ALPHARD: TOWARD A LANGUAGE TO SUPPORT STRUCTURED PROGRAMS

CARNEGIE-MELLON UNIVERSITY

PREPARED FOR
AIR FORCE OFFICE OF SCIENTIFIC RESEARCH
ADVANCED RESEARCH PROJECTS AGENCY

30 APRIL 1974

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

AD 785 417

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AFOSR - TR - 74 - 1434	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) ALPHARD: TOWARD A LANGUAGE TO SUPPORT STRUCTURED PROGRAMS		5. TYPE OF REPORT & PERIOD COVERED Interim 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) William A. Wulf		8. CONTRACT OR GRANT NUMBER(s) F44620-73-C-0074
9. PERFORMING ORGANIZATION NAME AND ADDRESS Carnegie-Mellon University Computer Science Department Pittsburgh, PA 15213		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 61101D A02466
11. CONTROLLING OFFICE NAME AND ADDRESS Defense Advanced Research Projects Agency 1400 Wilson Blvd Arlington, VA 22209		12. REPORT DATE April, 1974 13. NUMBER OF PAGES 20
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Air Force Office of Scientific Research /NM 1400 Wilson Blvd Arlington, VA 22209		15. SECURITY CLASS. (of this report) UNCLASSIFIED 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This report discusses the programming language tools needed to support the expression of "well-structured" programs. In particular it deals with the tools needed to express abstractions and their realizations; to this end it introduces the concept of a "form" to subsume the notions of type (mode), macro, procedure, generator, and coercion. An extended example is given together with the sketch of a proof of the example. The proof is included to support the contention that formal verification is substantially simplified when the abstractions and their realization are retained in the program text.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

ALPHARD:
Toward a Language to Support Structured Programs*

William A. Wulf
Computer Science Department
Carnegie-Mellon University
Pittsburgh, Pa.
April 30, 1974

Abstract

This report discusses the programming language tools needed to support the expression of 'well-structured' programs. In particular it deals with the tools needed to express abstractions and their realizations; to this end it introduces the concept of a 'form' to subsume the notions of type (mode), macro, procedure, generator, and coercion. An extended example is given together with the sketch of a proof of the example. The proof is included to support the contention that formal verification is substantially simplified when the abstractions and their realization are retained in the program text.

*This research was supported by the Advanced Research Projects Agency of the Office of the Secretary of Defense (F44620-73-C-0074) and is monitored by the Air Force Office of Scientific Research.

Introduction

In this paper I would like to present some ideas concerning the nature of the programming language facilities needed to support the construction of 'well-structured' programs. These ideas will be presented in the context of specific linguistic proposals for ALPHARD, a language being designed at Carnegie-Mellon University. A cautionary note is necessary however; ALPHARD is in its earliest stages of development, and the particular syntax used in the presentation is intended merely to be suggestive. Before beginning the presentation some introductory remarks on the language milieu are in order.

(1) Methodologies: 'Structured programming' and other methodological approaches to improving programs seem premature. While many of us might judge a particular program to be qualitatively 'better' than another, and might even be convinced that the difference has something to do with their respective 'structure', we do not yet have good characterizations of what we mean by 'structure' or what constitutes 'good' structure. Thus, for the moment at least, I would prefer to concentrate on programs rather than on the process of generating them. In particular I have chosen to focus on programming language issues as a vehicle for dealing with programs generically. That is, I view the development of tools, in this case a language, as a way of making the investigation of generic structural issues concrete and responsive to real-world issues. With that in mind, the following discussion of ALPHARD should be interpreted more as a view of what constitutes a well structured program than as a language proposal.

(2) Abstractions and Abstraction Tools: As Dijkstra has noted, abstraction is among our most powerful tools for reducing complexity. It shouldn't be surprising that program structure has something to do with abstraction, or that a proposal for language features to support structured programs should lean heavily on abstraction mechanisms. It should be noted, however, that abstractions come in two flavors: implicit and explicit. By implicit abstractions we mean those which are 'built in' and irrevocable - predefined data structures (array, set,...), storage allocation strategies (stacked, heap, static, controlled,...), and control relations (for, proc) are all examples of implicit abstractions. Explicit abstractions are those introduced by the programmer; a language supports such abstractions at the meta level in the sense that it provides mechanisms for defining the abstraction - procedures, macros, and (some) data structure and type (mode) definitions are the most common forms of these meta mechanisms.

It is this author's opinion that current languages contain far too many implicit abstractions, and far too few (or, at least, insufficiently general) mechanisms for defining explicit abstractions. The danger in implicit abstractions is twofold; in most current languages the implicit abstractions form an artificial lower barrier below which the programmer cannot descend, and in terms of which all 'higher level' abstractions must be expressed. This is both conceptually and technically inefficient. One need

only scan the literature on APL 'one-liners' to observe the consequences of a large number of implicit abstractions. We shall consider the nature of abstraction tools in greater detail below.

(3) Efficiency: I consider only one criterion for the efficiency of a higher-level language program to be acceptable: the code produced by the compiler for that language must be better than that produced by a competent assembly language programmer! To accept any less stringest requirement is to subjugate quality (reliability) to (usually invalid) efficiency considerations, and ultimately leads to poorly structured programs. The burden of meeting this requirement is shared by both the language design and the compiler. In particular, as far as the language design is concerned, it is crucial to avoid implicit abstractions which, either alone or especially interacting with other implicit abstractions, involve a distributed overhead for state maintenance.

(4) Proofs: I'm afraid that the current approach to proving the correctness of programs, e.g. the 'inductive assertion' method, is doomed to failure; yet the need for rigorously verified programs is paramount. Current methods essentially proceed from first principles for each program, and worse, re-prove the mathematics on which an algorithm is based in the process of proving a program which purports to implement the algorithm. We must, rather, devise methods which factor a proof along the same cleavage lines as the abstractions in a program. Moreover, we must be willing to accept the validity of programs whose relation to known, valid algorithms is transparent so long as the abstractions on which those programs are based are shown to be correct. In short, we must build a base of lemmas and theorems about existing abstraction realizations in terms of which programs utilizing those realizations can be verified relatively simply.* I will try to illustrate these remarks below.

(5) Data Structures and Sequencing Abstractions. In this section I would like to deal with two related issues - abstraction mechanisms for data structures and abstraction mechanisms for sequencing. In many ways the weakest aspect of abstraction mechanisms in current languages relates to data structures and their manipulation. With the exception of Simula [1], most languages provide only the ability to specify the (static) format of a structure; correlated manipulation of the structure and/or its elements is physically and conceptually separated from the structure definition. This point may be illustrated in many ways, but we shall focus on one - sequencing.

Most sequencing in a program is related to the data structures on which that program operates. Consider, for example, the following simple Algol '60 program:

---

*I believe it was Hamming who said something to the effect that computer scientists must learn to stand on each other's shoulders rather than on each other's toes.

```
begin
array A[0:N]; real S; integer i;
 ...
S := 0;
for i := 1 step 1 until N do S := S + A[i];
 ...
end;
```

Clearly in such a case the for clause is intimately related to the array A -- its intent is to step through A performing the statement 'S := S + A[i]' once, and only once, for each element of the index set. What we intended, but had no way to say in Algol, was:

```
forall a ∈ A do S := S+a;
```

Our inability to express ourselves this way in Algol has several unfortunate consequences:

- We were forced to say too much. For example, the order of the evaluation had to be specified when, in fact, it was immaterial.

- Changes are difficult. Any change in the representation of the conceptual entity denoted by A would require locating and altering the control used to sequence through A.

- Proofs are difficult. Although conceptually trivial, the formal proof of this simple loop using the inductive assertion method is not. At least in part the reason for this lies in the fact that the proof involves the 'dummy' control variable "i"; in part the difficulties arise because 'extraneous detail', e.g. the sequencing order, is explicit.

Some ALPHARD Ideas

In this section I would like to introduce some of the ideas in ALPHARD - at least as I currently perceive them. I do not intend to present the entire language, and I will rely heavily on suggestive examples and the reader's experience and good sense.

The only aspect of the language with which we shall deal in any detail is its abstraction mechanism(s). The goal is to explicate those aspects of the mechanisms which we feel are inadequately handled by existing language mechanisms. However, it should not be inferred that the mechanisms presented below are to be simply added to those of existing languages. Rather, we have attempted to define a single mechanism which subsumes the function of existing mechanisms. The extent to which this has been achieved is still unclear; however, if the attempt is found wanting, we would prefer to generalize it, or even replace it by a more suitable mechanism, than to accumulate related but disjoint mechanisms.

(1) Forms

The abstraction mechanism we shall introduce is called a form. A form may be thought of as something of a hybrid between a macro and a (Simula) class. Alternatively a form may be thought of in terms of its intended use - which is to subsume the concepts of type (mode), macro, procedure, generator, and/or coercion. Forms may be introduced by declarations such as:

```
form A(n,x) = {...};
```

where n and x are formal parameters of the form. Although we shall extend the definition of forms below, we can make several points here.

First, this declaration introduces the name A as the name of the particular form to the right of the equal sign. Second, all free names in the form, except those of the formal parameters, are bound at the declaration site. Third, the braces, "{ }", act somewhat like block delimiters (e.g. begin and end) with one major exception - certain names defined within the form may be "exported". That is, under conditions to be specified below, certain names declared within the form may be used outside it.

A form may be instantiated simply by mentioning its name together with any requisite actual parameters. One of the most common uses of instantiation is illustrated by the following:

```
form complex = {decl r:real,i:real; ...};
...
decl x:complex;
```

In this particular case the form 'complex' is being used as a type. This apparently trivial example actually raises several deeper issues:

- The symbol ':' should not be interpreted as an incidental part of the declaration syntax. It is, rather, a general binding operator which is useful in other than declarative contexts.

- The name 'real' is a form name, not a predefined type. In the strict interpretation there are no types in the language. However, it is reasonable to expect certain forms, e.g. 'real' and 'int', to be defined by a standard prelude.

Thus the declaration 'decl x:complex' introduces the name x and binds this name to an instantiation of the form complex. The instantiation of complex, in turn, forces the instantiation of two instances of the form 'real' and binds the names 'r' and 'i' to these two respectively. A more complete, and correct, explanation must await the discussion of 'extent' below.

(2) Names

Certain names may be 'exported' from an instantiated form in the sense that they are available as qualifiers to the name of the instantiated form. Thus, for example, if the following declarations have been made

```
form complex = {decl r:real,i:real;...;export r,i};
 ...
decl x:complex;
```

then the names 'x.r' and 'x.i' are valid - and, in this case, name the real variables representing the real and imaginary parts of x.

We have no particular prejudice about the syntax of qualification and therefore consider all the following to be equivalent, subject to the constraint that the style of qualification be uniform for each <name, qualification> pair*

x.i ≡ i(x) ≡ x(i) ≡ i[x] ≡ x[i]

We shall use one or another of these forms to suggest various intuitive meanings, and assume the programmer will do likewise.

In addition to the names which may be explicitly exported, certain names are implicitly exported from every form. These names correspond to actions which are implicitly triggered (called) because of the context in which the form is instantiated. It is difficult to expand this point until more has been said on other aspects of the language; be forewarned, however, that such names exist.

(3) Protection

We define each exported name of a form to be an 'access right' to instantiations of the form. Thus, in the example above, the use of the qualified name 'x.i' is viewed as an exercise of the right (privilege) to access the imaginary part of x.

In order to make the notion of an access right useful one must be able to specify permitted and/or required rights to an instantiated form. Although we must jump ahead of our story a bit to do so, consider:

```
form A = {... export a,b,c};
fcn F(x:A<a,b>) = <body>;
 ...
decl z:A;
 ...
[1] F(z<a>);
[2] F(z<a,c>);
[3] F(z<a,b>);
[4] F(z);
```

The intended interpretation is that the function F has a single parameter of 'type' A and that it requires 'a' and 'b' rights to that parameter -- that is, that it either uses the accesses 'x.a' and 'x.b', or that it calls some other function which, in turn, exercises these rights. At the call sites, lines [1]-[4], the actual parameters are qualified by the rights which the caller will allow the called routine to exercise; thus the calls at lines

---

*We also consider a(b)(c) ≡ a(b,c), thus x.i(z) ≡ x(i,z), etc.

[1] and [2] will not be allowed while that at [3] will be. If no rights qualification is specified, as in line [4], the allowed rights are defaulted to 'everything' available to the caller, thus the call at line [4] is also valid.

Rights qualification may also be attached to exported names. Thus a form may grant access to some of its internal variables but restrict the nature of such access. In the following example of the form complex only read access to the real and imaginary parts is granted.

```
form complex = {decl r:real,i:real;...export r<read>,i<read>};
```

(4) Extent

In many languages we refer to the 'extent' or 'lifetime' of a variable; in ALPHARD the term is given a somewhat more explicit meaning. However, in the initial part of the following discussion we would like to rely on the user's intuitive understanding of the term.

We shall allow declarations to specify an extent attribute, e.g.

```
decl own a:X;
```

The only extent attributes ultimately available are own and local, and if the extent attribute is omitted, local is defaulted. Within a form, however, two other extent attributes are permitted - common and unique.

The attribute unique implies: (1) that the declaration is unique to each instantiation of the form, and (2) that the extent of the declaration is identical to that of the instantiation. The attribute common implies: (1) that the declaration is common to all instantiations of the form, that is, shared between them, and (2) that the extent of the declaration 'covers' that of all instantiations of the form.

In those cases where a form is being used as a type, the quantities declared unique are those which are private to each variable of the specified type, those declared common are shared between all variables of the type. Thus, for example, one might implement the concept of a 'set' using linked lists as follows:

```
form set = {decl common p:pool, unique h:lhead;...};
...
    begin decl local s1:set;
    ...
        begin decl own s2:set;
        ...
        end;
    ...
    end;
```

In such a case the pool, 'p', is shared between all instantiations while there is a private lhead, 'h', for each of the instantiations. Since 'p' is shared between all

instantiations its extent must 'cover' them all - in this case it must be own because of the declaration of s2. The unique variables, on the other hand, have extents identical to the instantiations - local in the case of the 'h' associated with s1 and own in the case of the 'h' associated with s2.

Let us now return to a more precise characterization of the meaning of 'extent'. As noted in an earlier section some names are automatically exported from a form; four of these are initc, initu, finalc, and finalu. As with other automatically exported names, the semantics of ALPHARD specifies that the accesses ('operations' might be a more suggestive term in this context) represented by these names are automatically invoked in defined contexts. In this particular case the intent is that the operations defined by these names will perform initialization and finalization ('clean-up') actions on common and unique variables respectively.

The (only) meaning of the term 'extent' is the semantic rule governing the invocation of these operations! Although we shall not attempt a precise statement of this rule here, the intuition to be conveyed is that the invocation of the init and final actions of variables with own extent is to precede and follow the user-defined program actions, while these actions are invoked as part of block entry/exit for variables with local extent.

Strictly speaking, the concept of 'extent' has nothing to do with storage management. Storage management, rather, is explicit through an executable 'alloc' function. However, the (prelude) definition of such common forms as int, real, etc., is such that the conventional (Algol) correspondence between extent and storage allocation/deallocation is preserved.

(5) More on Forms

Earlier we introduced the notion of a form. It will be noted that subsequently we have used the notion almost synonymously with the conventional use of type or mode. This was in part due to an attempt to exploit the readers' intuitions, and in part due to an incomplete description of the notion. We would now like to expand the concept slightly. Consider an extension of our first example -

```
form A(n:int,x:int): y[z] = {decl u:y,v:z;...; assoc u[v]};
...
decl r[s]: A(3,4);
```

The ':', as noted earlier, is a general binding operator. In this particular case the declaration is intended to:

- introduce two names, 'r' and 's'
- 'r' and 's' are to be of type 'y' and 'z' respectively
- the names 'r' and 's' are to be associated with, or bound to, the variables 'u' and 'v' declared within the instantiated form. The assoc within the form establishes the association between the names to the left of the binding operator, ':', and those inside the form.

The type declarations of most languages, and the uses of forms which precede this example, introduce new conceptual entities, e.g. sets where none existed before. That is not the intent here at all. Rather, in this case, the form is being used to associate independent, existing entities (in this case a 'y' and a 'z') - and perhaps to introduce some additional operations. Thus, when instantiated, such forms do not create a single entity of a new type, and the combination of the assoc inside the form and the specification (e.g. ':y[z]') outside the form allow the user to bind names to each of these entities. The need for both the assoc and the specification may be seen in the following example.

```
form A(x:int):int = {decl unique v:vec(int,x),n:int;...assoc v[n]};
form B(x:int):vec[int] = {decl unique u:vec(int,x),m:int;...assoc u[m]};
...
decl a:A(5),b[c]:B(5);
```

Note that 'a' is associated with an element of 'v', 'b' is the vector 'u', and 'c' is the integer 'm'.

The use of square brackets in this example is pure syntactic sugar. Within the constraints imposed by possible ambiguity we wish to allow anything to the left of the ':', and to bind names tc the left of the colon positionally to the entities from the assoc in the instantiation of the form to the right of the colon. The previous examples of forms, e.g.

```
form A(n:int,x:int) = {...}
```

may be considered default instances

```
form A(n:int,x:int): A = {...}
```

This apparently trivial extension of the form syntax allows us to subsume the notion of literals, coercion (in the type-transfer sense), generators, and several other things. These are discussed briefly below.

(5.1) Literals

We view a literal as a variable with two special properties: its value does not change, and its value is suggested by its print (external) name. Given suitable definitions, all the following might be literals:

9	nine
IX	nine
4:15	quarter past four o'clock
blue	the color
NaCl	salt
Tuesday	the day

We view literals as being defined by forms. In some cases (Tuesday, blue) the definition may be simply a named form. In other cases the form that defines the literal

will provide a calculation that operates on the print name to produce the appropriate internal value.

(5.2) Type Conversion

We simply note in passing that type conversion requires either the ability to treat a single storage cell as being of more than one type or the application of a function to convert the representation. The assoc permits the former, the latter requires an explicit (named) function in Alphard.

(5.3) Access Functions

One of the major uses of forms will be to describe a conceptual data structure, its associated literals, operations, and accesses to its component pieces. A careful treatment of the access to elements of a conceptual data structure raises some deep issues concerning references and assignment which I prefer to avoid in a discussion at this level. However, I would like to note here that assoc is executable. Thus the general binding mechanism can be used to define access and sub-structuring operations (e.g. slicing).

(5.4) Generators

Earlier I discussed the need to relate control and data structures; now we have enough mechanism to illustrate the point. First let's consider a simple example:

```
form upto(f:int,t:int,b:int):int =
     {decl unique x:int;
     initu:: if (x←f) gtr t then signal;
     next::if (x←x+b) gtr t then signal;
     assoc x<read>
     };
...
forall i:upto (1,10,1) do S;
```

This example is intended to capture the simple stepping form of iteration control. Several things should be noted:

- The name 'next' is, like 'initu', one of those automatically exported names.

- Only read access has been granted to i in the statement S.

- forall, like decl, is a syntactic trigger to invoke one of these names. The forall construct, 'forall x:D do S', may be thought of as

```
begin
decl x:D;
until signal do (S; x.next);
end;
```

Note that the init function of D is invoked at instantiation, i.e. at the declaration of x. Thus the forall construct first initalizes the control variable, then alternately executes S and the 'next' function until termination is signaled.

This simple example doesn't illustrate the relation between data and control; however, consider the following representation of a set of integers in a vector:

```
form set(sz:int) =
{decl unique v:vec(int,sz), unique n:int;
initu::n←0;
 ...
form inset:int =
     {decl unique x:int;
     initu::if (x←1) gtr n then signal;
     next::if (x←x+1) gtr n then signal;
     assoc v[x]
     };
 ...
export inset
};
```

Then, if the declaration 'decl S:set(100)' has been made, the statement

```
forall v:inset(S) do v←v+1
```

will increment each element of the set.

There are two especially useful forms which we shall use below

```
forall D suchthat B do S
```

and

```
exists D suchthat B then S.1 else S.2
```

The first of these is the obvious extension to allow a test and is equivalent to

```
forall D do if B then S.
```

The second form will execute S.1 (precisely once) for the first case for which B is true and will execute S.2 only in the case that termination is signaled by D without B ever having been satisfied.

I consider this facility to be extremely important; for the first time I feel some confidence that all of the representational issues associated with a conceptual data structure may be isolated - thus making both changes and proofs incremental.

An Example

Although I have not dealt with all the language issues in ALPHARD, I hope that I have touched on enough of them that the reader's intuitions will carry him through the following example. The example is taken from the section on Data Structuring in [2] by C.A.R. Hoare. The problem is that of generating prime numbers using the sieve of Eratosthenes; Hoare states the problem as follows.

> Problem: Write a program to construct a set
>
> primes:powerset 2..N;
>
> containing all prime numbers in its base type. Use the method of Eratosthenes' sieve to avoid all multiplications and divisions.
>
> The method of Eratosthenes is first to put all numbers in the "sieve" and repeat the following until the sieve is empty:
>
> Select and remove the smallest number remaining in the sieve (necessarily a prime), and then step through the sieve, removing all multiples of that number.

After writing a nicely structured abstract version of the program, Hoare considers the constraint that the program be 'efficient' and, in particular, is not to use multiplications or divisions (except during initialization). The difficulty, of course, is that since the sets are represented essentially as bit vectors, he now cannot use division to determine the word and bit position corresponding to a particular integer. Instead he must use a pair of indices ('n.b' and 'n.w' below) to keep track of the word and bit positions. After some analysis he presents the following program:

```
primes, sieve:array 0..W of powerset 0..wordlength-1;
begin primefinder;
      n, next:(w,b:integer);
      for t:0..W do begin primes [t] := { };
            sieve [t] := range (0..wordlength-1)
            end;
      sieve[0] := {0,1};
      next.w := 0;
      while true do
            begin while sieve[next.w] = { } do
                  begin next.w := next.w+1;
                  if next.w > W then exit primefinder
                  end;
            next.b := min(sieve[next.w]);
            primes[next.w] := {next.b};
            n := next;
            while n.w≤ W do
                  begin sieve[n.w] := {n.b};
                  n.b := n.b + next.b;
                  n.w := n.w + next.w;
                  if n.b ≥ wordlength then
```

```
                        begin n.w := n.w + 1;
                        n.b := n.b - wordlength
                        end
                    end
                end
            end primefinder
```

While in some sense this program is 'well structured', it's a real shame that the abstractions leading to it have been lost. Moreover, because the realization of the abstractions aren't localized, but distributed throughout the text, any change in those realizations will require massive changes. I also claim that the proof of this program will be more difficult than in some sense it should be.

(Lest the reader think I'm criticizing this program, I'm not. I believe it represents one of the better examples of what can be done with existing abstraction tools. My criticism is of the lack of proper abstraction tools which, in turn, forces one to write this program in this way.)

Below I have written a (hopefully) equivalent version in ALPHARD. In writing this example I have written definitions 'top-down' - the implementation may, of course, require the most primitive things first. I have also hampered myself a bit so as not to go too far beyond the ALPHARD ideas presented earlier. For the same reason, the example is less efficient than it might be.

This implementation assumes the form 'word', a bit vector of convenient length for a particular underlying machine, has been predefined (e.g. in a 'standard prelude'). Specifically we assume that assignment to a word and access to individual bits is defined within this form.

```
begin

    decl sieve:iset(2,N,1), prime:iset(1,N,0);
    while not empty(sieve) do
            (include(prime,min(sieve)); removemults(sieve,min(sieve)));

    form iset(lb,ub,kv) =
        {decl unique b:powerset(lb,ub,kv);
            fcn include(x:b.inx) = b[x]←1;
            fcn removemults(x:b.inx) =
                    forall i:b.mults(x) do b[i] ← 0;
            fcn min:b.inx = b.min( );
            fcn empty:bool = b.empty( );
            export include,removemults,min,empty
        };

    form powerset(lb,ub,iv) =
        {decl unique p:vector(word,(ub-lb)/wordsize+1),max:pair(ub-lb);
            initu:: (forall x:invec(p) do
                    if iv = 0 then x←0 else x←-1;
                if iv=1 then forall x:upto(max.b+1,wordsize-1,1) do
```

```
                    p[max.w][x]←0);
            access:: [x:pair] = p[x.w][x.b];
            form inx:pair = {};
            fcn empty:bool = exists x:invec(p) suchthat
                                x ≠ 0 then false else true;
            fcn min:pair =
                begin decl m:pair(0);
                while p[m.w] = 0 do m.w ← m.w+1;
                while p[m.w][m.b] = 0 do m.b ← m.b+1;
                return m
                end;
            form mults(x:pair):pair =
                {decl unique t:pair(lb);
                    initu:: t←x;
                    next:: (t ← +(t,x); if >(t,max) then signal);
                    assoc t
                };
            export empty,min,mults,inx
        };

    form pair (iv) =
        {decl unique w,b:int;
            initu:: (w ← iv/wordsize+1;b ← iv mod wordsize);
            fcn +(a,b:pair):pair =
                begin
                decl c:pair(0);
                c.w ← a.w+b.w-1; c.b ← a.b+b.b;
                if c.b ≥ wordsize then (c.w ← c.w+1;
                                c.b ← c.b - wordsize);
                return c
                end;
            fcn >(a,b:pair):bool =
                begin
                if a.w > b.w then true else
                if a.w < b.w then false else a.b > b.b
                end;
            export w,b,+,>
        };

end
```

The reader will immediately recognize that the ALPHARD example is somewhat larger than Hoare's. The difference, however, arises because of the realization of abstractions (of powerset, for example) is explicit in the ALPHARD version. The explicit realization of these abstractions has a cost (size), but it also has advantages, e.g.:

- the realization may be changed
- proofs may be based on visible structure rather than implicit semantics (assumptions) of the language

In addition, one may assume that in practical environments a collection of useful realizations will accumulate - much like a subroutine library - eliminating the need for redundant definitions. Also, in a simple example such as this one the abstractions are sparsely used; in 'real' programs one expects to buy more notational 'leverage' from such definitions.

Proof

In this section I would like to sketch a proof of the previous program - the intent is not to present the proof in detail, but rather to provide sufficient detail so that it is convincing and credible. Inductive assertions are not included, but can be easily constructed by the interested reader. This proof is derived from one by London* for an earlier version of the same program.

First we assert that the top-level algorithm does not need verification - it is a simple transliteration of an algorithm whose valdity has been known for many years. Thus it only remains to show that the operations defined by the three major forms in fact accomplish the intended algorithm. The proof consists of a series of lemmas proceeding in bottom-up order (i.e., first with respect to 'pair', then 'powerset', and finally 'iset').

(1) Pair

Let A = A1*WS+A2 and B = B1*WS+B2, where 0 ≤ A2,B2<WS = wordsize. The pair init operation defines Pair[A] = (A1+1,A2) and Pair[B] = (B1+1,B2). Lemmas 1 and 2 show that arithmetic on pairs corresponds to ordinary arithmetic. These lemmas are used in verifying the powerset operations in Lemmas 5 and 6.

Lemma 1: Pair additon, denoted P+, preserves ordinary additon, i.e., Pair[A] P+ Pair[B] = Pair[A+B].

Proof: Pair[A] P+ Pair[B] = (A1+1,A2) P+ (B1+1,B2)
= (A1+B1+2-1,A2+B2)
= (A1+B1+1,A2+B2) where if A2+B2 ≥ WS,
then WS is subtracted from A2+B2 and
1 is added to A1+B1. (Note:
A2+B2<2*WS)
= Pair[(A1+B1)*WS+(A2+B2)]
= Pair[A+B]

Lemma 2: The pair operation, 'greater than', denoted P>, agrees with ordinary >, i.e. A > B iff Pair[A] P>Pair[B].

Proof: It suffices to consider three cases:
case 1, A1 > B1 (then A1 ≥ B1+1)
The P> operation returns true in this case, so we
must show A > B.

---

*private communication

$$A = A1*WS+A2 \geq A1*WS \geq (B1+1)*WS = B1*WS+WS > B1*WS+B2 = B$$

case 2, A1 = B1

$$A > B \text{ iff } A1*WS+A2 > B1*WS+B2 \text{ iff } A2 > B2$$

case 3, A1 < B1

This is verified by symmetry with the A1 > B1 case.

Note: Wordsize, WS, must be > 0; in particular it may be = 1.

(2) Powerset

Lemma 3: Assume ub-lb ≥ 0. The operation powerset initu initializes the powerset to all zeros or all ones according as iv = 0 or not.

Proof: Definition of initu, and assuming two's complement representation of 1. If iv = 1 it may be necessary to exclude part of the last element of the vector (non-empty because ub-lb ≥ 0. The second forall does this. Also observe that if the powerset fits exactly into the last element, then max.b+1 = WS > WS-1 and the second forall is executed zero times as required. It is assumed in the above that either iv=0 or iv=1 holds.

Lemma 4: The powerset predicate empty returns false iff the vector P contains a non-zero element.

Proof: Definition of empty.

Lemma 5: The form 'mults' defined in powerset, when used in the context of a forall, will produce valid pairs (indices into a specific instantiation of powerset) equivalent to an Algol-like

for i := n.1 **step** n.2 **until** **size-of-powerset** **do**

in whch addition and comparison are the relevant pair operations.

Proof: Definiton of forall, and Lemmas 1 and 2.

Lemma 6: Assume not empty(P). The operation min sets the pair M, which is locally declared in min to be (1,0) initially, to the pair

minimum ((W,B) such that (W,B) P≥ (1,0) and P[(W,B)] ≠ 0)

Proof: The first while statement finds

X = minimum (W such that W ≥ 1 and P[M.W] ≠ 0)

and sets M.W to X. The second while find

Y = minimum (B such that P[X][M.B] = P[M.W][M.B] ≠ 0)

and sets M.B to Y. Hence M is set to the pair (X,Y) as required.

The assumption not empty(P) assures that X and Y both exist. I.e., both while statements terminate "in bounds". The bounds are $1 \leq X \leq (UB-LB)/WS+1$ and $0 \leq Y \leq WS-1$.

(3) Iset

To include an element in an iset B at index N means B[N] := 1. Similarly, removing an element means B[N] := 0. Removemults(N) removes the elements at indexes N, 2N, 3N,..., size-of-powerset in view of Lemma 5. The min operation of iset uses min of powerset to return the minimum index of B with non-zero value (Lemma 6). Similarly empty of iset uses empty of powerset.

It remains to dispose of a detail of min in powerset (see Lemma 6). To discharge the assumption of Lemma 6, note that min is used only when "not empty(sieve)" holds in the top level while. Also note that the assumption is Lemma 3 (that iv of powerset is either 0 or 1) is satisfied iff kv=0 or 1.

As required, the initialization of prime (in the range 1 to N) is to the "empty" set (kv=0); the initialization of sieve (in the range 2 to N) is to the elements all being present (kv=1). $N \geq 2$ discharges the assumptions of Lemma 3 for both prime and sieve.

Conclusion

The intent of this paper has been to explore the nature of the language tools which seem to be needed in order to retain the abstractions and their realization in the text of a program. The notion of a form was introduced to do this along with explicit binding, extent, and protection control. We then attempted to show how forms may be used to define conceptual types, literals, access functions, coercion, and generators. We consider the concept of generators, which allow one to tie together data and control structures, to be especially important.

An example program for the sieve of Eratosthenes was presented to illustrate the mechanisms. A proof of this program was then sketched to illustrate how much proofs may model the program directly if the abstractions are retained in the program text. It seems especially significant that changes to the realization of one of the abstractions will impact only the proof of that realizaton!

Acknowlegments

Many people have contributed to the ideas reported here. My special thanks go to Ralph London, Mary Shaw, Dave Jefferson, Paul Hilfinger, Steve Hobbs, Gideon Ariely, Karla Martin, and Anita Jones. I am especially indebted to Ralph London for his original version of the proof, and for his corrections and amplifications to the current version.

References

[1] Dahl, Myhrhaug, Nygaard, "The Simula 67 Common Base Language," Norwegian Computing Centre, Oslo, 1968.

[2] Hoare, C.A.R., "Notes on Data Structuring," in Structured Programming, O. J. Dahl, E. W. Dijkstra, and C.A.R. Hoare (eds.), Academic Press, 1972, 127-138.